Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Automatic derivation of compact abstract syntax data types from concrete syntax descriptions

Florian Lorenzen

Bericht-Nr. 2011 – 11
ISSN 1436-9915

Automatic derivation of compact abstract syntax data types from concrete syntax descriptions

Florian Lorenzen

Technische Universität Berlin

florian.lorenzen@tu-berlin.de

No. 2011–11

ISSN 1436-9915

November 21, 2011

Abstract

We describe an algorithm (written in HASKELL) to automatically derive compact abstract syntax data types from concrete grammar descriptions. The input for the algorithm is almost the grammar language of the parser generator PAGE which not only constructs a parser but also the data types necessary to represent an abstract syntax tree. The algorithm of this report is suitable to minimize the data type used to represent the parsing result to both improve the handling of abstract syntax trees and their space requirements.

1 Description of the problem

Parser generators come in two flavours with respect to the result of a successful parsing of a given input:

- Some generators allow arbitrary semantic actions programmed in the host language to be inserted into the syntax description (e. g. YACC, HAPPY).
- Others derive a tree data type from the syntax description and generate the necessary code to manipulate values of this data type in the host language (e. g. PAGE).

In this report we are concerned with the latter type of parser generators since these generators have a practically annoying problem: the tree data types derived from a given grammar are usually much more verbose than a carefully handcrafted abstract syntax. The reason is that many non-terminals in the grammar are inserted for organisational or syntactical reasons like precedences or associtivities which are implicitly represented in a tree structure. But abstract syntax generated in a naive way contains all these unnecessary nodes.

In this report, we present an algorithm that distills a compact abstract syntax from a concrete syntax description. This algorithm can be used in a parser generator that also generates the abstract syntax data types to produce more manageable code.

G	$\rightarrow$	P `;` $\cdots$ P `;`	Grammar
P	$\rightarrow$	`Id ::=` R	Production
R	$\rightarrow$	`Id |` $\cdots$ `| Id`	Alternatives
	$\mid$	$S \cdots S$	Sequences
S	$\rightarrow$	`" string "`	Terminal symbol
	$\mid$	`id : " string "`	Terminal symbol with value
	$\mid$	`id :` N	Nonterminal symbol
N	$\rightarrow$	`Id`	Nonterminal
	$\mid$	`Id *` [`^ " string "`]	Repetition 0 or more times
	$\mid$	`Id +` [`^ " string "`]	Repetition 1 or more times

Figure 1: Concrete Syntax description language.

```
E  ::= E "+" E1          E   ::= Add | E1;
    |  E1;               E1  ::= Mul | E2;
E1 ::= E1 "*" E2         E2  ::= PE | Num;
    |  E2;               Add ::= lhs:E "+" rhs:E1;
E2 ::= "(" E ")"         Mul ::= lhs:E1 "*" rhs:E2;
    |  "NUM";            PE  ::= "(" pe:E ")";
                         Num ::= num:"NUM";
```

Figure 2: Grammar for arithmetic expression (left) and its equivalent in PaGe's input language (right).

We take the parser generator PaGe as example in the remainder of this report.

1.1 An example with PaGe

The input language of the parser generator PaGe [PG08, Hög09] differs from usual EBNF grammar description in two aspects:

- Productions with more than one right-hand side (sum productions) are restricted to single nonterminal right-hand sides.
- Nonterminal symbols and terminal symbols carrying a value (like identifiers or numbers) are explicitly named.

Figure 1 shows the productions for the input language used in this report. It differs from PaGe's input in not allowing optional parts in a production. We leave out this aspect but believe that it can be integrated easily. By the terminal symbols `id`, `Id`, `string` we mean identifiers beginning with a small letter, identifiers beginning with a capital letter, and arbitrary strings resp.

We illustrate the problem of unnecessarily large abstract syntax by the grammar of Fig. 2 for simple arithmetic expression, with associativity and precedence stated explicitly.

PaGe derives the following Java interfaces and classes from this grammar, where sum productions are mapped to interface and sequence productions are mapped to classes with corresponding attributes:

```
interface E {}
interface E1 extends E {}
interface E2 extends E1 {}
class Add implements E { E lhs; E1 rhs; }
class Mul implements E1 { E1 lhs; E2 rhs; }
class Num implements E2 { num:NUM; }
class PE implements E2 { E pe; }
```

If we were to write down the classes for the abstract syntax manually, we would perhaps write something like (see [AP02] where this style is used extensively)

```
abstract class E {}
class Add extends E { E lhs; E rhs; }
class Mul extends E { E lhs; E rhs; }
class Num extends E { num:NUM }
```

If we target a functional language like HASKELL we could use the following data type definition:

```
data E = Add {lhsAdd::E, rhsAdd::E}
       | Mul {lhsMul::E, rhsMul::E}
       | Num {num::NUM}
```

The JAVA and HASKELL representation are very similar in structure. We identify classes with summands and abstract classes with sum types.

To capture these similarities and to undertake a programming language independent development, we introduce an extra notation to specify data types for abstract syntax trees as shown in Fig. 3. This language is almost the same as the concrete syntax description language of Fig. 1 but allows more general right-hand sides and abstracts from the distinction of the repetition operators as well as labelled terminals/nonterminals.

We could easily generate the HASKELL or JAVA data types from the following specification:

```
E == Add* lhs:E rhs:E
  |  Mul* lhs:E rhs:E
  |  Num* num:NUM
```

The use of tags, i. e. the elements decorated by `*`, becomes now apparent: they identify constructors or classes. They may only appear as the first symbol in any right-hand side but we do not encode this restriction in the grammar of Fig. 3 because this leads to unnecessary clumsiness.

A	$\rightarrow$	$D \cdots D$	Collection of datatypes
D	$\rightarrow$	`str ==` V `|` $\cdots$ `|` V	Data type definition
V	$\rightarrow$	$F \cdots F$	Variant (sequence of fields)
F	$\rightarrow$	`str *`	Tag
	$\mid$	`str :` N	Labelled node
N	$\rightarrow$	`[` N `]`	Sequence of nodes
	$\mid$	`str`	(Non)terminal identifier

Figure 3: Language to describe data types for abstract syntax trees (`str` denotes arbitrary strings).

The task of our algorithm presented here is to derive the small description from the grammar.

We present the algorithm as a Haskell implementation in the next section. We interleave the code with extensive comments and illustrations.

The algorithm has been developed in an empirical process and we neither show any properties of it nor do we prove its correctness in any sense.

2 Algorithm

The algorithm is implemented in the module `GenerateASDT` and we begin by some formal noise and imports.

```
module GenerateASDT where

import qualified Data.Array as A
import qualified Data.Graph as G
import qualified Data.List as L
import qualified Data.Map as M
import Data.Maybe
import qualified Data.Set as S
```

The module is structured as follows:

1. We define data types and functions to handle the input language.
2. We define the data types to describe the abstract syntax, i. e. the result of our transformation.
3. We develop the algorithm step by step. We will use arithmetic expression as a running example to illustrate the different transformations.

4. We wrap up the individual steps in a function `generateASDT` at the end.

2.1 Input language

The abstract syntax of our input follows very closely the structure of the grammar in Fig. 1 but we do not distinguish between `id`, `Id`, and `string`.

```
type Grammar = [Prod]                                  -- G in Fig. 1

data Prod = Id ::= Rhs deriving (Eq, Show)             -- P in Fig. 1

data Rhs = Sum [Id]                                    -- R in Fig. 1 (alternatives)
         | Seq [Sym] deriving (Eq, Show)               -- (sequences)

data Sym = T String                                    -- S in Fig. 1 (terminal)
         | TV (Id, String)                             -- (terminal with value)
         | N (Id, NT) deriving (Eq, Show)              -- (nonterminal)
isT  s = case s of { T _  -> True; _ -> False }       -- Discriminators
isTV s = case s of { TV _ -> True; _ -> False }
isN  s = case s of { N _  -> True; _ -> False }

data NT = Sing Id                                      -- N in Fig. 1 (nonterminal)
        | Plus Id (Maybe String)                       -- (repetition 0 or more times)
        | Star Id (Maybe String)                       -- (repetition 1 or more times)
        deriving (Eq, Show)

type Id = String
```

Here is our running example of Fig. 2 in the Haskell encoding:

```
arithExpGrammar =
    [ "E"   ::= Sum ["Add", "E1"]
    , "E1"  ::= Sum ["Mul", "E2"]
    , "E2"  ::= Sum ["PE", "Num"]
    , "Add" ::= Seq [N ("lhs", Sing "E"), T "+", N ("rhs", Sing "E1")]
    , "Mul" ::= Seq [N ("lhs", Sing "E"), T "*", N ("rhs", Sing "E1")]
    , "PE"  ::= Seq [T "(", N ("pe", Sing "E"), T ")"]
    , "Num" ::= Seq [TV ("num", "NUM")]
    ]
```

2.2 Output data structures

Similarly to the input language we encode our output abstract syntax data type (ASDT) description in a few Haskell datatypes that closely follow Fig. 3. We omit the explicit definition of variants and encode them in lists of lists directly in `DT`.

```
type ASDT = [DT]

data DT = DT Id [[Field]] deriving (Show, Eq)  -- D and V of Fig. 3

data Field = Lab Id Node                       -- F of Fig. 3 (tag)
           | Tag Id deriving (Show, Eq, Ord)   -- (labelled node)

data Node = NSeq Node                          -- N of Fig. 3 (sequence)
          | TVal Id                            -- (terminal)
          | NVal Id deriving (Show, Eq, Ord)   -- (nonterminal)
```

We show the representation of the hand-crafted abstract syntax for arithmetic expression of Sec. 1.1 to illustrate the use of the data types:

```
arithExpAS =
    [DT "E" [ [Tag "Add", Lab "lhs" (NVal "E"), Lab "rhs" (NVal "E")]
            , [Tag "Mul", Lab "lhs" (NVal "E"), Lab "rhs" (NVal "E")]
            , [Tag "Num", Lab "num" (TVal "NUM")]
            ] ]
```

2.3 The ASDT generation algorithm

Step 1: Translating `Grammar` to `ASDT`

At first, we translate the grammar, basically as is, into the `ASDT` representation. Since simple terminals are irrelevant to the abstract syntax they are dropped in this transformation.

```
grammarToASDT :: Grammar -> ASDT
grammarToASDT = map prodToDT
```

Productions with several right-hand sides are are mapped to datatypes with singleton field sequence that contain nonterminals with empty labels.

Productions with one right-hand sides are converted to a sequence of fields each one labelled with the identifier from the grammar description.

```
prodToDT :: Prod -> DT

prodToDT (lhs ::= Sum ids) = DT lhs [ [Lab "" $ NVal id] | id <- ids ]

prodToDT (lhs ::= Seq syms) =
    DT lhs [[ symToField sym | sym <- syms, not $ isT sym ]]
    where
      symToField (N (id, Sing n))   = Lab id (NVal n)
      symToField (N (id, Plus n _)) = Lab id (NSeq (NVal n))
      symToField (N (id, Star n _)) = Lab id (NSeq (NVal n))
      symToField (TV (id, t))       = Lab id (TVal t)
```

Feeding `arithExpGrammar` to this function yields

```
E   == Add | E1
E1  == Mul | E2
E2  == PE | Num
Add == lhs:E rhs:E1
Mul == lhs:E1 rhs:E2
PE  == pe:E
Num == num:NUM
```

(We use the syntax of Fig. 3 instead of the more clumsy HASKELL representation here and in the following when showing examples and omit the "`:`" if the preceding identifier is empty.)

This still looks more or less like the grammar except that we have deleted plain terminal symbols.

Step 2: Expansion of single nonterminals

For the second step it is useful to have a map representation of the ASDT where each left-hand side of a data type is mapped to all its right-hand sides:

```
type ASDTMap = M.Map Id [[Field]]

asdtAsMap :: ASDT -> ASDTMap
asdtAsMap dts = M.fromList [ (lhs, rhs) | DT lhs rhs <- dts ]
```

We also need a "map of sets" in the following with the corresponding insertion function: if a key is not present in the map it is inserted mapping to a singleton set containing the value, otherwise the value is added to the key's set.

```
type MapOfSets k a = M.Map k (S.Set a)

mapOfSetsInsert k v m = if k `M.member` m
                        then let set = m M.! k
                             in M.insert k (S.insert v set) m
                        else M.insert k (S.singleton v) m
```

The goal of this step is to expand right-hand sides that only consist of a single nonterminal[1] to the defining right-hand side of that non-terminal. The rationale for this transformation is that we want to eliminate chain productions, especially in sum productions of the original grammar.

When we expand the defining right-hand side of a nonterminal into another right-hand side the original data type becomes useless because we can

[1] This excludes sequences like `l:[A]`.

use the data type of the expansion site to build that same node in an abstract syntax tree. We have to record which data types have become useless and which data types can be used instead. We set up an `ElimMap` for that purpose that maps left-hand sides of eliminated data types to the left-hand sides of expansion sites.

```
type ElimMap = MapOfSets Id Id
```

We now develop the basic function to perform the expansion. It takes the original ASDT (as a map, for convenience), an elimination map, and a single data type. It iterates over alternative right-hand sides of this data type (by a fold) and returns a new elimination map as well as new right-hand side alternatives.

The workhorse of this function is `subst` which replaces single nonterminals in an alternative by their definition. We have to avoid non-termination, i. e. given a definition

```
lhs == ... | id:lhs | ...
```

we must not insert the definition of `lhs` here.

There are three cases:

1. The definition of the single nonterminal is again a single nonterminal: we simply replace the nonterminal:

   ```
   lhs == ... | id:n | ...   ⇒ lhs == ... | id:n1 | ...
   n   == id1:n1
   ```

2. The definition of the single nonterminal is a single alternative. We insert an additional alternative at the expansion site, tagged with the left-hand side of the original definition:

   ```
   lhs == ... | id:n | ...
   n   == f1 ... fn
              ⇓
   lhs == ... | n* f1 ... fn | ...
   ```

3. The third possibility is that the definition of the single nonterminal consists of more than one alternative. In that case, we simply copy all the alternatives to our new right-hand side.

```
expand :: ASDTMap -> ElimMap -> DT -> (ElimMap, DT)
expand am elims (DT lhs alts) =
    let (alts1, elims1) = foldl subst ([], elims) alts
    in (elims1, DT lhs $ L.nub alts1)
    where
      -- Single nonterminal.
      subst (fields, elims) [Lab id (NVal n)] =
          if n == lhs then (fields, elims)  -- Avoid non termination.
          else case am M.! n of
                 -- Case 1.
                 [[Lab _ (NVal n1)]] -> ( fields ++ [[Lab id (NVal n1)]]
                                        , mapOfSetsInsert n lhs elims)
                 -- Case 2.
                 [x]                 -> ( fields ++ [Tag n : x]
                                        , mapOfSetsInsert n lhs elims)
                 -- Case 3.
                 x                   -> ( fields ++ x
                                        , elims)

      -- Other field.
      subst (fields, elims) x = (fields ++ [x], elims)
```

We extend the function to expand a single data type to expand a sequence of data types:

```
expandAllDTs :: ASDTMap -> (ElimMap, ASDT) -> (ElimMap, ASDT)
expandAllDTs am (elim, asdt) = L.mapAccumL (expand am) elim asdt
```

Expansion is an iterative process, so we take the fixed point of this function:

```
expandASDT :: ASDT -> (ElimMap, ASDT)
expandASDT asdt = let am = asdtAsMap asdt
                  in fix (expandAllDTs am) (M.empty, asdt)


fix :: Eq a => (a -> a) -> a -> a
fix f x0 = let x1 = f x0
           in if x1 == x0 then x0 else fix f x1
```

Step 3: Delete eliminated data type definitions

We can now delete all data types from our ASDT that have been eliminated, i. e. fully expanded on some other right-hand side.

```
pruneASDT :: ElimMap -> ASDT -> ASDT
pruneASDT elims asdt = filter notEliminated asdt
    where
      elimLhs                  = M.keysSet elims
      notEliminated (DT lhs _) = S.notMember lhs elimLhs
```

Step 4: Identify and delete redundant data types

After step 3, the data type for our running example looks like this:

```
E  == Add* lhs:E rhs:E1
    | Mul* lhs:E1 rhs:E2
    | Num* num:NUM
E1 == Mul* lhs:E1 rhs:E2
    | Add* lhs:E rhs:E1
    | Num* num:NUM
E2 == Add* lhs:E rhs:E1
    | Mul* lhs:E1 rhs:E2
    | Num* num:NUM
```

We do not have useless sums anymore but instead identical datatypes `E1`, `E2`, and `E2`. We now eliminate these.

In this step, we order the right-hand sides of all data types by a partial order and extract the largest elements of this set. We introduce some defintions to work with partial orders first: a data type representing the result of a "partial" comparison and a typeclass with a method for partial comparison:

```
data PartialOrdering = PLT | PEQ | PGT | PUnrelated
    deriving (Eq, Show)

class PartialOrd a where
    partialCompare :: a -> a -> PartialOrdering
```

We are interested in comparing right-hand sides of a data type definition. For this purpose, we consider a right-hand side as a set of lists and give a general instances for partially ordered sets by set inclusion:

```
instance Ord a => PartialOrd (S.Set a) where
    partialCompare x y | x == y             = PEQ
                       | x `S.isSubsetOf` y = PLT
                       | y `S.isSubsetOf` x = PGT
                       | otherwise          = PUnrelated
```

(The total order on the set's elements is necessary for the set implementation in use.)

We illustrate the idea of this step by an artificial example. Suppose, the outcome of step 3 of our algorithm is

```
D == Q* F [F]
E == P* E F
   | Q* F [F]
   | R* D E E
F == R* D E E
   | Q* F [F]
```

Whenever we need a node of type `D` we can take `F` or `E` since they both have the `Q*` variant. Analogously, whenever we need a node of type `F` we can take `E`. We always want to use the "largest" data type possible because this enables us to discard all smaller one. Using the partial order for right-hand defined above, we can order the three data types as

$$\texttt{D} \sqsubseteq \texttt{F} \sqsubseteq \texttt{E}$$

(`x` $\sqsubseteq$ `y` means that `partialCompare` returns `PLT` or `PEQ`).

From this ordering we extract the largest element `E` and replace all occurences of smaller elements with `E`. The smaller elements can now be deleted from the ASDT. For our artificial example, we end up with:[2]

```
E == P* E E
  |  Q* E [E]
  |  R* E E E
```

We have neglected one important issue so far: since right-hand sides are only ordered partially there might be several largest elements and smaller elements might be related to more than one of the largest. To illustrate this situation, we add the definition

```
G == Q* F [F]
  |  S* G E G
```

to the previous example. As the Hasse diagram in Fig. 4 shows we can either use `G` or `E` to replace `D`. But we cannot eliminate `G` in favor of `E` because they are unrelated. The solution is to keep `E` and `G` and to to consistently chose any of them to replace `D`. Choosing `G` for `D` gives us the reduced data types

```
E == P* E E
   | Q* E [E]
   | R* G E E
G == Q* E [E]
   | S* G E G
```

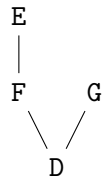

Figure 4: Hasse diagram of the partial order of the `D`, `E`, `F`, `G` example.

The implementation of this step proceeds as follows:

1. Build the directed graph representing the partial order.
2. Extract all vertices with indegree of one. Those are the largest data types (they are only comparable with themselves).
3. Delete all data types from the ASDT that were not detected previously.

The function `asdtRhsToSets` converts the list of right-hand sides to a set of right-hand sides:

```
asdtRhsToSets :: ASDT -> [(Id, S.Set [Field])]
asdtRhsToSets asdt = [ (lhs, S.fromList rhs) | DT lhs rhs <- asdt ]
```

To build the graph, we set up an adjacency map that maps left-hand sides to their neighbors. A neighbor is a left-hand side that has a smaller right-hand side.

```
adjacencyMap :: [(Id, S.Set [Field])] -> MapOfSets Id Id
adjacencyMap dts = foldl insertIntoAdj M.empty pairs
    where
      pairs = [ (dt1, dt2) | (dt1, dts1) <- zip dts (init $ L.tails dts)
                           , dt2 <- dts1 ]

      insertIntoAdj adj ((lhs1, rhs1), (lhs2, rhs2)) =
          case partialCompare rhs1 rhs2 of
            PUnrelated -> adj
            PGT        -> mapOfSetsInsert lhs1 lhs2 adj
            PEQ        -> mapOfSetsInsert lhs1 lhs2 adj
            PLT        -> mapOfSetsInsert lhs2 lhs1 adj
```

The function `asdtToGraph` uses the `Data.Graph` library to build the graph of the partial order.

[2]We lose some type safety of course but that is usually a price we have to pay for smaller data types.

```
type POGraph = (G.Graph, G.Vertex -> (Id, Id, [Id]), Id -> Maybe G.Vertex)

asdtToGraph :: ASDT -> POGraph
asdtToGraph asdt =
    let adjMap  = adjacencyMap $ asdtRhsToSets asdt
        adjList = M.toList adjMap
    in G.graphFromEdges [ (k, k, S.toList v) | (k, v) <- adjList ]
```

We now extract the largest data types, i. e. those with an indegree of one.

```
largestDTs :: POGraph -> S.Set Id
largestDTs (g, vf, kf) = S.fromList [ lhs | vertex <- vertices
                                          , let (_, lhs, _) = vf vertex
                                          , indegree A.! vertex == 1 ]
    where
      indegree = G.indegree g
      vertices = G.vertices g
```

We are now in position to delete all data types from our ASDT that are not large enough.

```
deleteDTs :: S.Set Id -> ASDT -> ASDT
deleteDTs largest asdt =
    [ dt | dt@(DT lhs _) <- asdt, lhs `S.member` largest ]
```

The function `deleteRedundantDTs` encapsulates the entire step and returns the new ASDT, the graph of the partial order, and the set of largest data types.

```
deleteRedundantDTs :: ASDT -> (ASDT, POGraph, S.Set Id)
deleteRedundantDTs asdt = let g       = asdtToGraph asdt
                              largest = largestDTs g
                              asdt1   = deleteDTs largest asdt
                          in (asdt1, g, largest)
```

Step 5: Update remaining data types

Since we deleted data types in the previous two steps the remaining ones now use undefined data types. So, we have to replace them appropriately, either by the left-hand side of their expansion site (if they were deleted in step 3) or by a larger data type (if they were deleted in step 4).

As a preparation step, we first have to calculate all data types that the remaining ones are able to represent. That is, we calculate all nodes reachable by the largest data types in the graph of the partial order.

```
smallerDTs :: POGraph -> S.Set Id -> [(Id, [Id])]
smallerDTs (g, vf, kf) largest =
    [ (lhs, dts) | lhs <- S.toList largest
                 , let key = fromJust $ kf lhs
                 , let vertices = G.reachable g key
                 , let dts = [ dt | vertex <- vertices
                                  , let (_, dt, _) = vf vertex ] ]
```

In fact, we need the inverse of the above mapping because we want to find a data type able to represent some deleted data type. The function `largerDTs` invertes this mapping:

```
largerDTs :: [(Id, [Id])] -> ElimMap
largerDTs mapping = foldl (flip $ uncurry mapOfSetsInsert) M.empty
                      [ (s1, l) | (l, s) <- mapping, s1 <- s ]
```

To perform the replacement step, we need the elimination map from step 2 as well as the output of `largerDTs` with the set of largest data types.

The replacement is basically a traversal of all data types down to the individual nodes. Once at a node (in function `replaceNode`), it is checked if the node belongs to the largest data types. If so, it is left unchanged. If not, we check if there is a known larger data type that we can take instead. If the node is not in `larger` it has already been eliminated in step 2 and we extract its replacement from `elims`. The replacement may have been deleted in step 4, therefore it is subject to `replaceNode` again to find the appropriate larger data type.

`larger` as well as `elims` are maps of sets, i. e. the replacement is not unique. The function `repFrom` extracts some arbitrary but fixed, i. e. fixed for each key, element from the set. This choice may most likely be improved.

```
replaceDTs :: ASDT -> ElimMap -> ElimMap -> S.Set Id -> ASDT
replaceDTs asdt elims larger largest = map replaceDT asdt
    where
      replaceDT (DT lhs rhs) = DT lhs (map replaceAlts rhs)
      replaceAlts fields     = map replaceField fields

      replaceField (Lab id n) = Lab id $ replaceNode elims larger largest n
      replaceField field      = field


replaceNode :: ElimMap -> ElimMap -> S.Set Id -> Node -> Node
replaceNode elims larger largest (NSeq n) =
    NSeq (replaceNode elims larger largest n)
```

```
replaceNode elims larger largest (NVal n)
    | n `S.member` largest = NVal n
    | n `M.member` larger  = NVal $ repFrom larger n
    | n `M.member` elims   = replaceNode elims larger largest
                               $ NVal (repFrom elims n)

replaceNode _ _ _ n = n


repFrom :: MapOfSets Id Id -> Id -> Id
repFrom reps id = head $ S.elems (reps M.! id)
```

We wrap up the entire step in function `updateDTs`:

```
updateDTs ::
    ASDT -> POGraph -> S.Set Id -> ElimMap -> (ASDT, MapOfSets Id Id)
updateDTs asdt g largest elims =
    let smaller = smallerDTs g largest
        larger  = largerDTs smaller
        asdt1   = replaceDTs asdt elims larger largest
    in (asdt1, larger)
```

Feeding our example grammar for arithmetic expression through steps 1–5, we obtain

```
E == Add* lhs:E rhs:E
   | Mul* lhs:E rhs:E
   | Num* num:NUM
```

This is the result we were looking for! Nevertheless, some grammars still produce artefacts, which are removed by the last step 6.

Step 6: Remove duplicate right-hand sides

It sometimes happens that step 5 returns a data type with right-hand sides like

```
E == ...
   | P* E E
   | Q* E E
   | ...
```

which might or might not be the desired outcome. Basically, if `P` and `Q` have been insserted in the original grammar for semantic reasons, i. e. they denote different things, the result is perfectly alright. But if `P` and `Q` have been inserted for syntactical reasons, i. e. for disambiguation or precedences one of them will be sufficient.

The last step trys to eliminate duplicates by relating the replacement of data types back to the original grammar. If the productions become

identical by the replacement, we consider them syntactical and duplicates are removed. This step has been developed empirically and may turn out to be nonsense in certain circumstances.

The function `removeDuplicateDTs` scans all data types and removed duplicates on the right-hand side using a special equality function `dup`. This function extracts the original production from the grammar and replaces all nonterminal symbols in that production by their counterparts in the final ASDT (using the function `replaceNode` defined in step 5).

```
removeDuplicateDTs ::
    ElimMap -> ElimMap ->  S.Set Id -> Grammar -> ASDT -> ASDT
removeDuplicateDTs elims larger largest grammar asdt =
    let grammarMap = M.fromList [ (lhs, rhs) | lhs ::= rhs <- grammar ]
    in [ DT lhs rhs1 | DT lhs rhs <- asdt
                     , let rhs1 = L.nubBy (dup grammarMap) rhs ]
    where
      dup grm (Tag tx:xs) (Tag ty:ys) = let px = grm M.! tx
                                            py = grm M.! ty
                                            upx = updProd px
                                            upy = updProd py
                                        in upx == upy

      updProd (Seq syms) = [ updSym sym | sym <- syms ]

      updSym s@(T _)              = s
      updSym s@(TV _)             = s
      updSym (N (id, Sing n))     = N (id, Sing $ replacement n)
      updSym (N (id, Plus n sep)) = N (id, Plus (replacement n) sep)
      updSym (N (id, Star n sep)) = N (id, Star (replacement n) sep)

      replacement n =
          let NVal n1 = replaceNode elims larger largest (NVal n) in n1
```

2.4 Wrapping it up

The function `generateASDT` composes the individual steps 1–6:

```
generateASDT grammar =
    let asdt1               = grammarToASDT grammar
        (elims, asdt2)      = expandASDT asdt1
        asdt3               = pruneASDT elims asdt2
        (asdt4, g, largest) = deleteRedundantDTs asdt3
        (asdt5, larger)     = updateDTs asdt4 g largest elims
        asdt6               = removeDuplicateDTs elims larger
                                  largest grammar asdt5
    in asdt6
```

3 A more complex example

We illustrate the technique by a more complex example: λ-expression with let-bindings. The following grammar fully specifies binding strength, associativity, and precedence of λ-expressions in the usual way:

```
E  ::= A | B | F | L;
A  ::= H | B1;
H  ::= I | F1 | L1;
B  ::= P | Q;
F  ::= "\" id:"ID" "." body:E;
B1 ::= "(" b:B ")";
F1 ::= "(" f:F ")";
L1 ::= "(" l:L ")";
P  ::= fun:H arg:A;
Q  ::= fun:B arg:A;
I  ::= id:"ID";
L  ::= "let" defs:D+^";" "in" body:E;
D  ::= lhs:"ID" "=" rhs:E;
```

Most nonterminals have been introduced for purely syntactic reasons and PAGE produces four interfaces and nine classes. Feeding the grammar into the algorithm of Sec. 2 yields

```
E == I* id:ID
   | F* id:ID body:E
   | L* defs:[D] body:E
   | P* fun:E arg:E
D == lhs:ID rhs:E
```

which is as compact as a handwritten abstract syntax.

4 Open issues

There remain a few open issues and problems with the algorithm presented.

- It has been developed empirically by an iterative process of inspection of examples and modifications of the algorithm. For this reason, there might be grammars for which the output is inadequate.
- The algorithm does not properly record which data elements must be used by a parser when performing reductions. Part of this information

is encoded in the maps `larger` and `elims` but this does not include which variant of a data type has to be used.

- The fifth step updates nonterminals in a rather ad-hoc way. It should be possible for the user to annotate which nonterminals should remain in the abstract syntax data types and which are present merely for technical reasons. With this information, the function `repFrom` will probably work much more predictable.
- Optional parts in productions have been omitted entirely.
- The artifical example used in the description of step 4 of the algorithm keeps two identical variants with tag `Q*` but in two different data types. We are not sure if this can (or should) be eliminated or if this case shows up in practice at all.

References

[AP02] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[Hög09] Christoph Höger. Generation of incremental parsers for modern IDEs. In Jens Knoop and Adrian Prantl, editors, *KPS'09: 15. Kolloquium Programmiersprachen und Grandlagen der Programmierung*, 2009.

[PG08] Peter Pepper and Martin Grabmüller. Compiler generator für Opal-2, April 2008. Lecture slides Compilerbau Projekt 2 SS 2008.